Artificial Intelligence Laboratory
Massachusetts Institute of Technology

AI Memo 898 April 1986

Discovery Systems

Kenneth W. Haase Jr.

ABSTRACT

Cyrano is a thoughtful reimplementation of Lenat's controversial Eurisko program, designed to perform automated discovery and concept formation in a variety of technical fields. The "thought" in the reimplementation has come from several directions: an appeal to *basic principles*, which led to identifying constraints of modularity and consistency on the design of discovery sytems; an appeal to *transparency*, which led to collapsing more and more of the control structure into the representation; and an appeal to *accountablity*, which led to the explicit specification of dependencies in the concept formation process.

The process of reimplementing Lenat's work has already revealed several insights into the nature of Eurisko-like systems in general; these insights are incorporated into the design of Cyrano. Foremost among these new insights is the characterization of Eurisko-like systems (which I call *inquisitive* systems) as search processes which dynamically reconfigure their search space by the formation of new concepts and representations. This insight reveals requirements for modularity and "consistency" in the definition of new concepts and representations.

This report describe research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505 and in part by the Office of Naval Research under Office of Naval Research contract N00014-79-C-0260.

1 Introduction

Cyrano is a thoughtful reimplementation of Eurisko [Len83a] developed over the last year at the MIT AI lab. Like Lenat's controversial program, Cyrano is designed to discover new concepts and representations in a variety of technical domains. For example, from an initial definition of set theoretic operations, the program synthesizes the concept of numbers and various operations on numbers.

While the development of Cyrano is still in progress, early results have produced insights into the design and performance of discovery systems in general. Discovery is so fundamental a process that any success requires fundamental explanations. Lenat and Brown in [LB83] propose that discovery systems succeed due to a close connection between syntax and semantics in their representation. I believe that this is only part of the story. In this article I describe four additional insights into the nature and design of discovery systems.

- In Section 2, I show how **Eurisko-like systems may be viewed as search processes which reconfigure their own search space.** I call these sorts of processes *inquisitive processes*: processes which extend — during search — the conceptual vocabulary in which their search is cast. Inquisitive processes are contrasted with acquisitive processes which acquire new descriptions by search or instantiation in a representational space whose form is initially — and invariably — fixed. This view is a significant extension of Lenat's analysis in [Len76,Len82].
- In Section 3, I explain why **concept formation in discovery systems must be functionally modular.** Since the progress of an inquisitive process is driven by concepts formed (or provided) at earlier moments of the process, the inputs and outputs of each moment of the process must be explicitly accessible to the preceding and succeeding moments of the process. This requirement for explicitness demands that the formation of new concepts be a **module** characterized by its explicit inputs and outputs. To illustrate this requirement, several non-modular parts of AM are detailed and criticized.
- In Section 4, I argue that **the formation of new concepts must be "consistent" as well as modular.** Given that a concept formation

module has — on the basis of experimentation and empirical analysis — produced an array of extended concepts, these generated concepts must be amenable to further experimentation and extension by the same module. Informally, the inputs and outputs of the module must "talk about" the same sorts of things. Many of AM's most powerful heuristics were crippled by this lack of consistency.

- In Section 5, I describe why **an inquisitive process must ultimately be introspective.** As the conceptual vocabulary of the process grows, the empirical performance of its search and concept formation engine will decline. The process must ultimately reflect on, modify, and extend this engine if it is to proceed effectively past this point. This reflection is a property the design of Cyrano shares with Eurisko.

Finally, in Section 6, I describe the integration of these insights into the design of Cyrano. Among the highlights of this design are: a uniform representation of concepts in a subsumption lattice of types; a general representation of empirical regularities, structured around the confirmation process; and a control structure based on the organization of tasks into experimentally determined classes.

2 Inquisitive Exploration

The operation of a system like AM or Eurisko is often described as a heuristic search through a space of operators and concepts; in this framework, the power of the program arises from the effectiveness of the generating and pruning heuristics for this search space. As with any search process, it is critical that the representation of the search space expose the constraints of the domain. In particular, the representation syntax and the represented semantics must be closely coupled, ensuring that small syntactic variations (steps in the search space) produce meaningful (or even interesting) semantic definitions. I call this coupling of syntax to semantics the "tightness" of the representation.

But the search space of operators and concepts is described in terms of the operators and concepts themselves, a representational vocabulary which is being constantly extended by the ongoing search process. For each generation or cycle of the discovery process, the conceptual vocabulary of its search is determined by concepts formed in the preceding generations. To succeed,

the concept formation mechanism must maintain the tightness of its representation over indefinitely many generations of discovery and abstraction. The consistency of this concept formation process is at least as critical as any semantic-syntactic tightness in the starting representation.

Tightness of representation — a property critical to any heuristic search — is the principle to which Lenat and Brown [LB83] attribute the success of AM and Eurisko. AM succeeded, they argue, because the language in which AM represented its mathematical concepts was LISP, which was designed — at its roots — from a mathematical basis. Small changes in the LISP definitions of simple recursive functions produced simple and meaningful changes in the mathematical structures they defined or operated upon. While this analysis is correct, it is incomplete; the credit for the success of AM (and the blame for its eventual malaise) is at least as much due to its mechanisms for extending its conceptual vocabulary and representational search space.

The novelty and power of inquisitive systems lies in this careful incremental evolution of representations and definitions. The significance of AM is not that it discovered multiplication, but that it defined numbers; not that it generated the operation Divisors-Of, but that it defined primes and noticed they were interesting. From AM's initial configuration, the first definition of multiplication was only one syntactic step away, and Divisors-Of only five. An exhaustive search would have eventually found them. What led to these discoveries and marked them as interesting were the concepts which AM defined along its path to them. These generated concepts allowed AM to focus sufficiently to make (and find interesting) its later discoveries and allowed us (or particularly, Lenat) to relate AM's derivations to recognized mathematical concepts. The semantically transparent and tight representation space provided for mathematics by LISP is certainly important, but at least equally important are the mechanisms which extend that space *while preserving its transparence and tightness.*

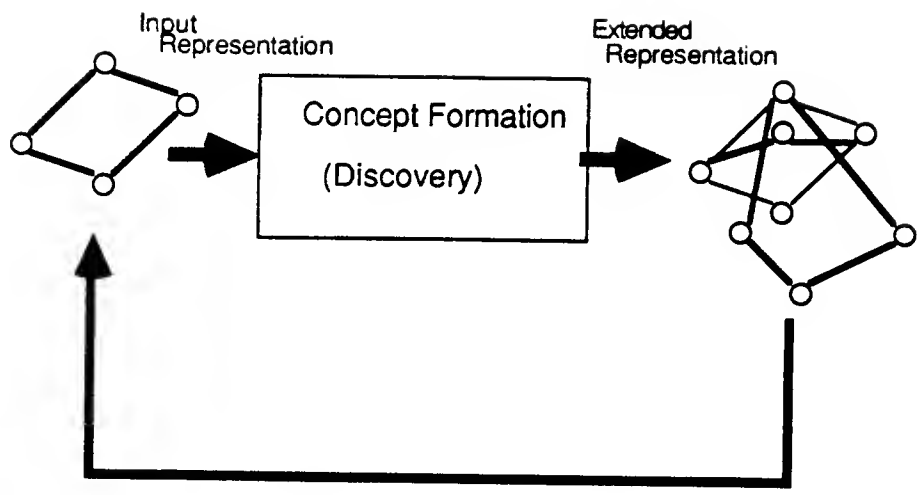

Figure 1. Discovery can be profitably viewed as a cycle of representational extension; concepts formed in one cycle of discovery are used as terms in the vocabulary given to the next.

3 Modularizing Concept Formation

The first step to tightness is transparence and the first step to transparence is explicitness; this yields a constraint on the form of an inquisitive process which we describe here as the **modularity constraint.** The insight of this section is that the formation of new concepts must be a **module** with clearly and explicitly defined inputs and outputs. The inputs are an experimental vocabulary and a way of generating (or referencing) its empirical behavior; the outputs are new concepts and representations (new vocabulary) which capture or exploit certain empirical properties of the inputs (regularities, coincidences, etc). Because each cycle of an inquisitive process builds on the representations (the results) of the cycles before it, the output of each cycle must be accessible as input to the next. This requirement of the discovery cycle is pictured in Figure . In order for the connection between output definitions and input representations to be realized, the output of the formation process must be **explicit.**

An example of modular concept formation is the following heuristic from AM, Eurisko, and Cyrano:

If Some (but not most) examples of an interesting class C are also examples of an interesting class D, and D is not already a specialization of C,

Then Define and study a specialization of C and D which is the intersection of C and D.

This heuristic catches one particular sort of empirical regularity: coincidental overlap of classes. The class it produces is accessible, even to the same heuristic, for further analysis and specialization. Further, the concepts it defines actually enhance the explicitness of the representational search space by separating off and declaring possibly interesting cases of predicate/property intersection. Modular extensions like this maintain the transparency of the search space from cycle to cycle in the inquisitive process.

An example of non-modular concept formation is the CANONIZE operation (or heuristic) of AM. The CANONIZE operation takes two related two-place predicates (one is a generalization of the other over the same domain) and produces an automorphism of their domain which preserves the algebraic structure they define over it. Precisely, given $p : A \times A \Longrightarrow \{T, F\}$ and a generalization $r : A \times A \Longrightarrow \{T, F\}; p(x, y) \rightarrow r(x, y)$, CANONIZE finds a function $f : A \Longrightarrow A$ such that $r(x, y) \longleftrightarrow p(f(x), f(y))$. This function f generates a "canonical representation" of A which preserves the equivalence partition defined over A by r. In generating f, CANONIZE recognizes the algebraic structure of A under r and exploits it, but the partition of A is never explicitly and accessibly declared.

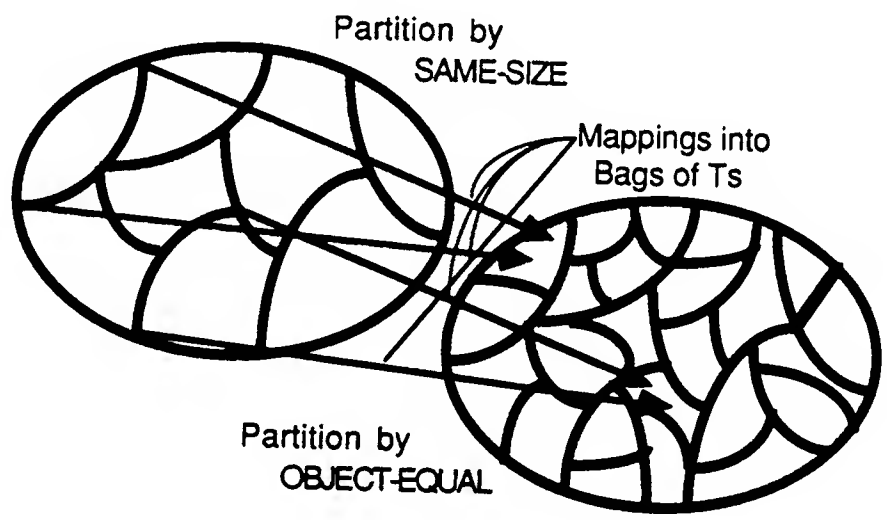

Figure 2 . AM's CANONIZE heuristic found a partition preserving mapping from the domain of SAME-SIZE to the domain of OBJECT-EQUAL. This mapping transformed each element of SAME-SIZE lists into the unique symbol T, producing OBJECT-EQUAL lists.

CANONIZE plays a critical role in AM's progress, defining the canonicalization of bags (multisets) under the SAME-SIZE relation (cardinality) relative to LIST-EQUAL. Given a synthesized notion of SAME-SIZE (a generaliza-

tion of LIST-EQUAL), AM tried to find a mapping of lists into lists such that lists of the same size would be mapped into lists that were equal. The successful result of this attempt was a mapping (f) which took every element of a list and replaced it with the single symbol `T`. `BAGS-OF-Ts`, the range of this mapping (representing the equivalence partitions of SAME-SIZE), was interesting because of where it came from and was later renamed `Numbers` by Lenat. This one discovery, depicted in Figure , was the basis of AM's forays into number theory, where all of its more significant discoveries were made.

CANONIZE is an instance non-modular concept formation because AM never explicitly constructed the partition of the set of bags, but merely exploited its structure to produce the canonicalization f. The recognition of p and r as equivalence relations is never explicitly declared; if it had been, it would be available for confirmation, identification, or exploitation by either later phases of the inquisitive process or a human user interacting with the program. These properties are buried inside the CANONIZE heuristic and never see the light of accessibility by later phases of concept formation and analysis.

In Cyrano, the recognition of structures like equivalence classes is noted explicitly; the class of bags is specialized into the set of bags *qua* algebraic group (i.e. the subset of bags over which same-size is an equivalence relation), and then this is specialized into its disjoint equivalence classes. These equivalence classes then become the objects of new operations 'raised' from the class the relation was originally defined over.

Equivalence partitions are only one of a broad class of structural properties which Cyrano looks for in its empirical observations; these broad empirical classes are axes of concept formation which support the *consistency* of the concept formation process. This consistency is demanded by the second constraint on concept formation in an inquisitive process: *the consistency constraint.*

4 Consistent Construction

The modularity requirement arises from the structure of inquisitive processes in general: the "discovery cycle" which grinds experiment into representation must close on itself. A "semantic" version of the modularity constraint

is the **consistency requirement** placed on the inputs and outputs of a concept formation module.

The experiments performed and the patterns looked for in concept formation are determined by the "sensibility space" and the "interest space" of the input representation. These are specified (of necessity) syntactically and, as pointed out in [LB83], the success of the formation process (to which I will append "at any moment or generation of the inquisitive process") depends critically on the tightness with which these syntactic specifications match the actual space of sensible or interesting constructions.

The consistency constraint arises from this tightness requirement; the concept formation process should preserve — in the new representational vocabulary it generates — the tightness of the original syntactic specification. Since this tightness arises from the representations recognized by the inputs, the forms produced by the outputs of the formation module must be *consistent* with the forms recognized and exploited by its inputs. This requirement is a constraint placed on both the inputs and the outputs of a concept formation module: informally, they must talk about the same sorts of things.

Many of the heuristics in AM and Eurisko satisfy the consistency constraint. The class coincidence heuristic mentioned above, for instance, deals with arbitrary classes and produces a class which refines already established regularities and — at need — may be analyzed and further specialized by the same or other heuristics. AM's operation restriction heuristic also satisfies the consistency constraint:

> **If** The domain D of an interesting operation O has an interesting specialization C,
>
> **Then** Define and study O' which is the operation O restricted to Cs.

The new operation this defines can be analyzed by the same heuristics which found the original O to be interesting and further extended on the basis of this analysis. In one instance, AM used this heuristic to study addition restricted to primes, leading to the proposal of Goldbach's conjecture (that any even number may be expressed as the sum of two primes).

On the other hand, AM's CANONIZE heuristic — to criticize it once more — violates the consistency constraint. We will recall that CANONIZE recognizes an relative algebraic property of two relations over their common domain and produces a canonicalization of the domain to itself which preserves

this property. But this description invests CANONIZE with more generality than it deserves; CANONIZE actually recognized only a handful of particular equivalence partitions defined over the set of list structures by various structural mutations such as element variance, permutation or deletion. In this, CANONIZE violates the consistency constraint because its outputs — the fixed points of simple structural mutations in a space defined by the two predicates — are distinctly separate from the forms recognized by its inputs (arbitrary structures). Put simply, structural canonicalization immediately obsoletes itself.

A more general version of CANONIZE — modularly working off of generally recognized and explicitly declared equivalence partitions — could define a canonicalization by selecting distinguished elements from each partition and defining that as a canonical set. Or more generally, it could define the set of equivalence partions as a class of its own with operations which are defined in terms of operations on the objects partitioned. Such a version of CANONIZE would satisfy both the modularity and consistency constraints we have formulated. We can imagine this more general (and more modular) version of CANONIZE eventually examining (and finding structure in) synthesized notions like vectors (lists of numbers) or dot-products, once it had defined numbers. But AM's clumsy and impoverished CANONIZE was impotent once its objects moved beyond simple structures to numbers, a class it had itself defined.

Lenat recognized that the primary reason for AM's eventual malaise was a particular violation of the consistency constraint: AM's concepts outgrew its heuristics. His solution, proposed in [Len76] and implemented in Eurisko [Len83a], was to make the inquisitive process itself — heuristically defined — a domain for discovery and evolution by meta-heuristics. Instead of making consistency a constraint on the initial design of the concept formation engine, consistency was to be dynamically maintained by a battery of evolutionary meta-heuristics.

In Cyrano, we have instead chosen to implement the consistency constraint directly, having the program always operate with a vocabulary of functions, operators, and classes. The concept formation module extends this vocabulary by recognizing and acting on certain highly exploitable domain independent regularities — called "concept germs" by Minsky [Min86] and

"cognitive cliches" by Chapman [Cha83] — to which are attached batteries of reasoning, problem solving, and exploration/experimentation heuristics. The outputs of the concept formation module are concepts and functions reflecting these regularities and therefore exploitable — in virtue of their batteries of attached heuristics — by the next cycle of the inquisitive process. By choosing experiments based on these regularities and forming new concepts around them, the consistency constraint is embedded in the concept formation module of Cyrano.

In the final analysis, this principle also emerged from Eurisko's development, as Lenat's meta-heuristics began to express the same sort of domain independent properties incorporated into Cyrano's design. The meta-heuristics presented in [Len83b] capture the same sort of domain independent properties as concept germs or cognitive cliches. Lenat identifies these concept formation principles or heuristics as methods significantly more specific than weak methods like "Generate and Test" but still far more general than domain specific methods like "Try the choke." It is not surprising that these methods emerged from Eurisko's development; they are the result of designing around the consistency constraint to find principles prevailing over many domains or many generations of an inquisitive process.

From Eurisko's eventual convergence with the consistency constraint, it would appear that Lenat's original reply to the AM's consistency crisis — having meta-heuristics dynamically maintain consistency — failed and was superseded by embedding the consistency constraint in the domain independent formation heuristics of the program. This is true insofar as the only metric of consistency of represenation was the overall performance of the heuristics using it. But such a blanket condemnation of Lenat's solution is unfair. Even when consistency has been built into the structure of the inquisitive process, ensuring that it continues to run, meta-heuristics still play a pivotal role in the inquisitive process, ensuring that it continues to succeed.

5 Inquisitive Introspection

The twin constraints of modularity and consistency maintain and constrain the evolving representational search space of an inquisitive process. But this space is still enormous and ever growing; an inquisitive process must

choose one path of representational experimentation from among many possibilities. If we don't want this choice to be arbitrary, the inquisitive process must become a heuristic search.

This characteristic of inquisitive processes has been an implicit bias in previous sections, whose examples were taken from three heuristic discovery programs: AM, Eurisko, and Cyrano. Taken alone, the constraints of modularity and consistency describe a representational space which — in theory — could be enumeratively searched; but most of the nodes reached in such a search would be — while syntactically plausible and apparently suitable for further exploration — dead-ends of little or no utility to the program in the future. Empirically, no examples or patterns will be found and the program's labors in the direction would be wasted.

We would like our heuristic search to avoid such short term dead-ends. But the sorts of paths which are successful in any given domain or generation of discovery (the paths we would like our heuristics to select) are particular to that domain or generation; in order to maintain the effectiveness of its search, the inquisitive process must modify — or appropriately extend — the heuristic engine by which it proceeds along paths constrained by consistency.

This process of modification or extension could be managed by a special separate process (working independently of the inquisitive process itself), but it seems more sensible (and ultimately more powerful) to manage this modification by turning the inquisitive process on itself. Eurisko and Cyrano (AM was not introspective) work in this way, turning its own performance into a domain for empirical experimentation and examination. In Cyrano, the heuristics that drive the inquisitive process become operators in a space of tasks and concepts, and these operations and tasks are analyzed, organized, modified, and specialized based on their empirical performance. The new concepts and operations developed in this domain specify new heuristics, specialized or synthesized for particular domains or new representations.

But in describing this introspection we intrude on the structure of the inquisitive program itself, a matter of design and implementation rather than of theoretical properties. Having so intruded, we shall complete our step and begin a description of Cyrano's particular implementation.

6 Cyrano: The Implementation

The principles above arose from a careful study of reports on AM and Eurisko, scattered conversations with Lenat himself, some hours interacting with Eurisko at Xerox PARC, and — most importantly — a prototypical implementation of Cyrano which duplicated about half of AM's reported performance. This prototypical implementation duplicated most of the control structure and representation of Eurisko and based on its development, the principles above were formed and clarified. A program implementing these principles — the latest version of Cyrano — is still under development, but enough of its design has been specified to sketch its critical components and new innovations.

Cyrano is implemented in Scheme and was developed under UNIX on Hewlett-Packard Bobcat machines and on Symbolics 3600's using a SCHEME to Common LISP translator developed by the author. SCHEME was chosen for reasons of elegance, simplicity, and transportability.

The implementation of Cyrano revolves around a subsumption lattice of types and classes. All of Cyrano's concepts are defined as nodes in this lattice and Cyrano's experiments and observations are all described by relations embedded in or attached to the lattice. New concepts are well-formed extensions to the lattice which then are amenable to further experimenation and extension. Cyrano's control structure is also organized around the lattice, which places the program's activities and projects in particular classes within the lattice. This organization replaces the priority queues of AM and Eurisko with myriad "focus classes" of related activities.

These components of Cyrano's design remain essentially experimental; they may be abandoned or changed as their actual performance or behaviour is revealed. Most derive from experience with Eurisko and the prototypical Cyrano: the reasoning behind each design decision is presented below.

6.1 The Type Lattice

All of Cyrano's concepts are represented as "types" in a lattice of generalization and specialization. A fragment of this lattice is shown in Figure . All of Cyrano's discoveries about the properties of its representations are described in this lattice and the program's new definitions consist of additions

to the lattice. Many of Cyrano's actions begin with *classifying* some object in this lattice and using the resulting classification to determine some set of actions. Some of the types in the lattice are the natural classes of various domains; others are empirically collected sets of objects or types (which are also objects); but most are *analytic types* which combine other types into new definitions.

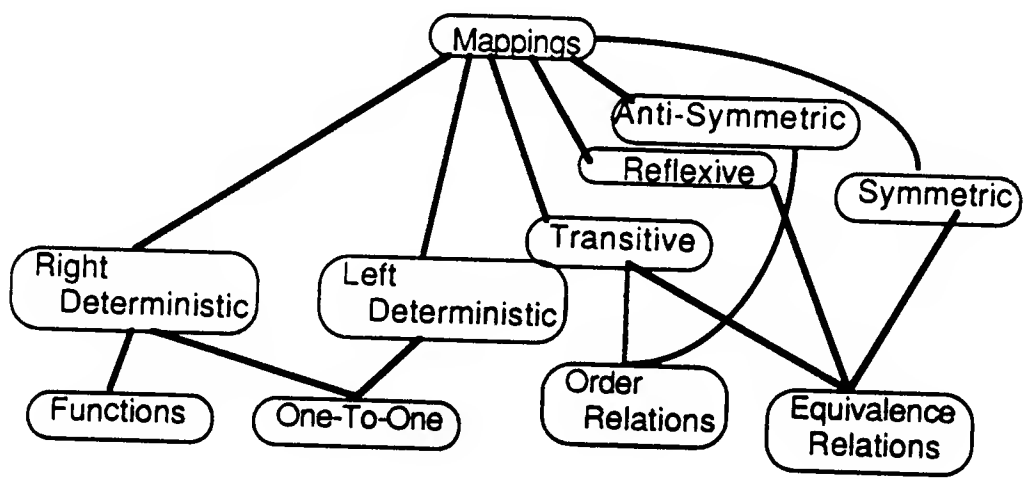

Figure 3. All of Cyrano's concepts and meta-concepts are uniformly represented in a lattice of types.

Some of the types represented in the lattice are *composite*: they specify types of tuples satisfying particular element or inter-element constraints. Functions and relations, for instance, are represented as pairs of other objects and the fact that a function has a particular LISP implementation is merely a heuristic for finding examples of such object pairs. This generality is an attempt to move Cyrano beyond completely specified domains into areas where examples are not always effectively enumerable, such as the real world!

Types in the lattice are of two basic sorts: analytic and synthetic. Analytic types are types whose definition is solely in terms of other types; for instance, the intersection or union of two established types or a constraint on some component of a composite structure. Synthetic types are types whose definition is provided by the "world": for instance, enumerated sets, LISP predicates, or user defined classes. One important sort of synthetic type is the *empirical class* which I describe below; it defines and implements Cyrano's notion of regularities and experimentation.

New types are defined in a combinator language in terms of either existing types or — in the case of some synthetic types — in terms of the behaviour

some external interface. The following are examples of type definitions:

```
;;; Defining a type by intersecting two existing types.
(define bachelors (type-intersection men unmarried))
;;; Defining a type by merging two existing types.
(define agents (type-union humans intelligent-programs))
;;; This defines a function call by saying that the image
;;;  of the function CAR for function calls is function names.
(define function-calls
  (image-constraint CAR function-names))
;;; This defines points in 3-space as a cross product of
;;;  reals.
(define points (cross-product reals reals reals))
;;; This defines the class of LISP functions in a particular
;;;  implementation; its uses the LISP predicate functionp.
(define functions (simple-type functionp))
;;; This defines chord triads which the user says are harmonious.
(define harmony
 (type-intersection
  (query-type "harmonious?")
  (cross-product notes notes notes)))
```

Definitions like these describe both Cyrano's initial domain and the constructed domains it develops over time; the program's defining actions construct new types and place these types — as objects — in appropriate classes in the lattice. The central process in Cyrano generates examples for classes in the lattice, in turn triggering the definition of new classes for which examples must be found.

Each of the combinators above possesses a type-inference procedure for computing — on creation — the types neccesarily above and below it in the lattice. One of the properties of the lattice is that for any two given types, no new subsumption relations will ever be established between them. Any newly created type will have new subsumption relations, but those new relations will never posit new relations between types already in the lattice.

It has been shown that type inference in a distributive lattice with complementation is NP-complete[BL84]; my lattice implementation gets around

this intractability by weakening representing complementation to representing a limited form of *disjointness*. Complementation — information that any object not in one class **IS** in another — allows the inference of certain subsumption relations not otherwise computable from the lattice; this asymmetry makes subsumption in complementary lattices an NP-complete problem. Information about disjointness (as limited for Cyrano) only enables further inferences about disjointness, a weaker inference. This weakening is sufficient to make the type inference problem tractable, though still enabling many useful inferences.

Using the lattice to represent all the program's knowledge — both provided and defined — ensures that the modularity constraint is satisfied. Each cycle of concept definition produces extensions to the lattice and places these extensions in meta-classes in the lattice; all of this information is then available to the next phase of discovery.

6.2 Empirical Classes

An inquisitive process proceeds by a cycle of recognition and definition: empirical regularities noticed in one domain vocabulary are used to define terms in the domain vocabulary of the next cycle. The recognition of regularities is one key component of any inquisitive process. In Cyrano, all regularities are represented by subsumption relations in the lattice of concepts.

In particular, Cyrano defines an *empirical class* as a class of classes whose members accidentally (i.e. not by definition) satisfy some empirical property. An empirical class K is determined by two functions: a test function K_{Test} and a confirmation function $K_{Confirm}$. These functions translate an individual class into a *test class* and a *confirmation class*. A regularity K is true of a class c if all instances of $K_{Test}(c)$ are also instances of $K_{Confirm}(c)$ (e.g. for all known examples: $K_{Test}(c) \subseteq K_{Confirm}(c)$). Figure illustrates a view of confirmation classes as overlapping sets.

The simplest empirical classes have a constant

$$K_{Confirm} = C$$

and

$$K_{Test}(x) = x.$$

These classes encode the simple regularity that some set is contained in a particular constant set. "All X's are red" or "R is a reflexive relation" are

instances of such classes. More complicated empirical classes generate test and confirmation classes which combine instances of the class they are testing. For instance, the regularity "F preserves R" has a test class of "pairs of Fs whose inputs are related by R" and a confirmation class of "pairs of Fs whose outputs are related by R."

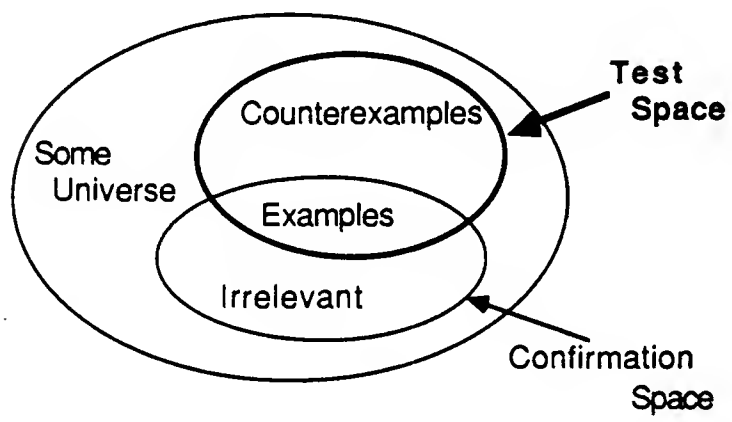

Figure 4 . Empirical classes describe empirical regularities by potential accidental subsumption/subset relations in the lattice.

Membership in an empirical class K is determined by defining a class of examples $K_{Test}(c) \wedge K_{Confirm}(c)$ and a class of counterexamples $K_{Test}(c) \wedge \neg K_{Confirm}(c)$. Each of these has classification daemons which — when a counterexample or some quota of positive erexamples is discovered — assert the membership of the class being tested in either the appropriate empirical class or its complement. The representation of all empirical regularities in this manner is an attempt to satisfy the consistency constraint; any defined concept is likely to be amenable to this level of raw empirical analysis. It depends only on definitions of sets and tuples, rather than on arbitrary properties of mutatable list structures.

Of course, setting up a test situation is only part of the confirmation process; it is also neccesary to generate the examples which will fall into the situation. When a test situation is created, a *task* is created for generating examples of $K_{Test}(c)$. The scheduling of these tasks, again using the type lattice, is introduced below.

Cyrano's control structure has two components: a set of *classification daemons* and a set of active *projects*. A classification daemon is a procedure run on new examples of particular types; a project is an activity divided into quanta of action.

Classification daemons work as follows. Whenever a new potentially interesting object is found or generated, its terminal types (the most specific types it satisfies) are collected. The lattice is then climbed — in the generalization direction — from this set, and at each type along its ascent the classification daemons of that type are applied to the object. This process is called *classification* and is Cyrano's fundamental action.

When a new definition is generated by Cyrano, the definer gives it some set of properties and classifies it. This classification triggers daemons which — based on the properties of the definition — propose hypotheses in terms of empirical classes. These hypotheses set up the confirmation machinery described in the previous section, which then waits for confirming or disconfirming examples. When a hypothesis is confirmed or disconfirmed, it is given the appropriate empricial property (added to the appropriate class) and classified again. This classification may produce either new hypotheses or new definitions, which will once more turn the crank of analysis and definition.

Cyrano's discovery activities divide classification daemons into two interleaved control phases: recognition and extension. In recognition, empirical regularities in an input vocabulary are recognized; in extension, new representations are defined based on these noted regularities. Recognition and extension are further divisible into two control stages each: recognition begins by the *hypothesis* of possible regularities and continues to the *confirmation* of these regularities through analysis by empirical classes; extension begins by definition of primitive concepts and proceeds to their elaboration by pragmatic example-generation information, or the definition of associated operations and functions. These phases, while conceptually distinct, are interleaved into the classification process. Classifying a definition produces hypotheses which set up confirmation machinery; classifying a definition with some noticed empirical property produces new definitions which captialize on that property. Classification of examples and counterexamples drives the confirmation machinery set up by previous classifications of definitions; classification of new definitions produces auxiliary definitions which elaborate simple constructions.

Despite this complexity of actions and triggers, classification daemons are a more or less passive mechanism, reacting to some source of examples streaming in from the world. Cyrano applied to analyzing a mass of scientific data might function precisely in this mode. On the other hand, in many domains (perhaps eventually in all) Cyrano may have to seek or generate examples. For this purpose, AM and Eurisko's notion of *tasks* has been partially appropriated.

Eurisko's tasks were organized into several separate agendas and ordered by a universal priority within each agenda. At any moment, Eurisko worked on a single agenda, selecting and executing the highest priority task on the agenda. Cyrano abandons Eurisko's priority mechanism, choosing instead to enrich the agenda structure. At any point, Cyrano is working on a class of tasks — its *focus class* — which are related in some way. When focussing on a class of tasks, Cyrano executes all the tasks in the class, either in some order or at random. Over time, Cyrano observes the empirical properties of these classes, defining new classes of tasks to which the focus may eventually shift.

Task execution occurs in three stages: triage, execution, and post mortem. In *triage*, the classifier runs on the task description, and daemons construct an "implementation" for the task. This implementation is then used in an *execution* of the task. After the task completes, it is classified again as a *post mortem* perhaps triggering changes in focus or new defintions of task types. Task execution may be thought of as a generator for examples of actions, as well as a mechanism for acting.

Unlike AM and Eurisko's tasks, Cyrano's tasks are never completed in one shot. Instead, they describe ongoing processes divided into quantized actions. The one shot actions of Eurisko are replaced by simple procedure calls, generally from the execution of a classification daemon.

Currently, Cyrano's tasks are only example generation tasks; in the future they will become the backbone of evolving problem solvers for the domains Cyrano is learning in.

7 Conclusion

In conclusion, I will restate the insights into discovery systems put forth in this paper:

- **Eurisko-like systems may be viewed as search processes which reconfigure their own search space.**
- **Concept formation in discovery systems must be functionally modular.**
- **The formation of new concepts must be "consistent" as well as modular.**
- **An inquisitive process must ultimately be introspective.**

The implementation described here is in active progress, and the results in six months will offer new insights on the mechanisms described in the last section. In the same way that the prototpyical implementation of Cyrano produced the insights above, I look forward to the next phase of implementation as fertile ground for newer insights.

8 References

[BL84] Ronald J. Brachman and Hector J. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In *AAAI-84*, American Association for Artificial Intelligence, 1984.

[Cha83] David Chapman. *Naive Problem Solving and Naive Mathematics.* Working Paper 249, MIT Artificial Intelligence Laboratory, 1983.

[LB83] Douglas B. Lenat and Jon S. Brown. Why AM and Eurisko Appear to Work. *Artificial Intelligence*, 23, 1983.

[Len76] Douglas B. Lenat. *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.* PhD thesis, Stanford University, 1976.

[Len82] Douglas B. Lenat. AM: Discovery in Mathematics as Heuristic Search. In Douglas B. Lenat and Randall Davis, editors, *Knowledge Based Systems in Artificial Intelligence*, McGraw-Hill Book Company, 1982. Several appendices of examples were trimmed from the original version of the thesis in this book version.

[Len83a] Douglas B. Lenat. Eurisko: A program which learns new heuristics and domain concepts. *Artificial Intelligence*, 21, 1983.

[Len83b] Douglas B. Lenat. Theory Formation by Heuristic Search. *Artificial Intelligence*, 21, 1983.

[Min86] Marvin Minsky. *The Society of Mind.* Simon and Schuster, 1986. Forthcoming.